



Магические методы в ООП



Table 1-1. Special method names (operators excluded)

Category	Method names
String/bytes representation	<code>__repr__</code> , <code>__str__</code> , <code>__format__</code> , <code>__bytes__</code>
Conversion to number	<code>__abs__</code> , <code>__bool__</code> , <code>__complex__</code> , <code>__int__</code> , <code>__float__</code> , <code>__hash__</code> , <code>__index__</code>
Emulating collections	<code>__len__</code> , <code>__getitem__</code> , <code>__setitem__</code> , <code>__delitem__</code> , <code>__contains__</code>
Iteration	<code>__iter__</code> , <code>__reversed__</code> , <code>__next__</code>
Emulating callables	<code>__call__</code>
Context management	<code>__enter__</code> , <code>__exit__</code>
Instance creation and destruction	<code>__new__</code> , <code>__init__</code> , <code>__del__</code>
Attribute management	<code>__getattr__</code> , <code>__getattribute__</code> , <code>__setattr__</code> , <code>__delattr__</code> , <code>__dir__</code>
Attribute descriptors	<code>__get__</code> , <code>__set__</code> , <code>__delete__</code>
Class services	<code>__prepare__</code> , <code>__instancecheck__</code> , <code>__subclasscheck__</code>

Методы конструирования и инициализации

`__new__` **`__init__`**

```
class Person(object):  
    instance = None  
    def __new__(cls, *args, **kwargs):  
        if not cls.instance:  
            cls.instance = object.__new__(cls)  
        return cls.instance  
    def __init__(self):  
        self.__name = 'Peter I'  
  
obj_one = Person()  
obj_two = Person()  
print( obj_one is obj_two )
```

Метод **`__new__`** вызывается первым. Он открывает пространство памяти затем вызывается метод **`__init__`** .
Используется для переопределения immutable классов (int, str, tuple)



Метод документирования `__doc__`

```
# дандер для документирования класса и методов
# Комментарии нужно помещать сразу после объявления
class Test(object):
    ''' Класс Test для демонстрации '''

    def show(self):
        ''' This is Show Function DocString '''
        pass

t = Test()
print(t.__doc__)          # Описание класса
print(Test.__doc__)       # Описание класса
print(t.show.__doc__)     # Описание описания метода
```

Строковые методы `__str__` `__repr__`

Дандеры для отображения объекта в виде строки вызываются при работе с функциями **`print()`** **`str()`**

```
class Car:
    def __init__(self, model, color, vin):
        self.model = model
        self.color = color
        self.VIN = vin

    def __str__(self):
        return f"Модель: { self.model} с VIN номером {self.VIN}"

    def __repr__(self):
        return f"Модель: { self.model} с VIN номером {self.VIN}"
```

```
car = Car("Mercedes", "red", "WDB12443345J081498")
print(car)
print(str(car))
```

А что будет если нет строковых дандеров ?



Строковые методы `__str__` `__repr__`

```
class Car:
    def __init__(self, model, color, vin):
        self.model = model
        self.color = color
        self.VIN = vin
```

```
car = Car("Mercedes", "red", "WDB12443345J081498")
```

```
print(car)
<__main__.Car object at 0x7fe009b78a60>
```

```
print(str(car))
<__main__.Car object at 0x7fe009b78a60>
```

1. Для преобразования строк в классах можно использовать дандер методы `__str__` и `__repr__`
2. В свои классы всегда следует добавлять метод `__repr__`.



Магические методы сравнения

`__eq__(self, other)`

Определяет поведение оператора равенства `==`

`__ne__(self, other)`

Определяет поведение оператора неравенства, `!=`

`__lt__(self, other)`

Определяет поведение оператора меньше, `<`

`__gt__(self, other)`

Определяет поведение оператора больше, `>`

`__le__(self, other)`

Определяет поведение оператора меньше или равно, `<=`

`__ge__(self, other)`

Определяет поведение оператора больше или равно, `>=`

Откуда такие обозначения ?

```
#!/bin/bash
#This Script accepts a positive integer from the user
#and calculate its factorial
#Date: Sep 2015
if [ $# -ne 1 ]; then
    echo "Error: One Argument Expected"
    exit 3
else
    if [ "$1" -eq "$1" 2> /dev/null ]; then
        if [ $1 -ge 0 ]; then
            result=1
            for i in `seq $1`
            do
                let "result*=$i"
            done
            echo "Factorial of $1 equals $result"
        else
            echo "You Should Enter a Positive Integer"
            exit 2
        fi
    else
        echo "You Should Enter an Integer"
        exit 1
    fi
fi
```

Равенство значений объектов класса

`__eq__`

```
class Car:
    def __init__(self, model, color, vin):
        self.model = model
        self.color = color
        self.VIN = vin

    def __eq__(self, obj):
        if not isinstance(obj, Car):
            raise ValueError("Передан другой тип
                               объекта")
        return (self.VIN == obj.VIN)
```

```
car_one = Car("Mercedes-benz", "silver", "WDB1240221J081498")
car_two = Car("Mercedes-benz", "red", "WDB1240221J081498")
print(car_one == car_two)  ← Что вернет сравнение ?
```

Что нужно для полного сравнения ?

Оператор больше `__gt__`

```
class Car:
    def __init__(self, model, price):
        self.model = model
        self.price = price

    def __gt__(self, obj):
        if not isinstance(obj, Car):
            raise ValueError("Передан другой тип объекта")
        return (self.price > obj.price )

    def __str__(self):
        return f"Модель: {self.model} с ценой {self.price} "

car_one = Car("Mercedes-benz", "50000000")
car_two = Car("Aurus Senat", "50000000")
print(car_one > car_two) < ?
print(car_one < car_two) < Что если изменить знак ?
```

Оператор меньше `__lt__`

```
class Car:
    def __init__(self, model, price):
        self.model = model
        self.price = price

    def __lt__(self, obj):
        if not isinstance(obj, Car):
            raise ValueError("Передан другой тип объекта")
        return (self.price < obj.price )

    def __str__(self):
        return f"Модель: {self.model} с ценой {self.price} "

car_one = Car("Mercedes-benz", "50000000")
car_two = Car("Aurus Senat", "50000000")
print(car_one > car_two) ← ?
```



Название класса `__class__`

```
class Car:
    def __init__(self, model, price):
        self.model = model
        self.price = price

    def __str__(self):
        return f"Модель: {self.model} с ценой  
{self.price} "
```



```
print(Car.__class__)

obj = car("Lada", "800000")

print(obj.__class__)    ?
```




`__dict__` словарь для хранения атрибутов

```
class Car():
    color = "Red"
    def __init__(self, model, price):
        self.model = model
        self.price = price

    def __str__(self):
        return f"Модель: {self.model} с ценой {self.price} "

obj = Car("Mercedes-benz", 5_000_000)
print(obj.__dict__)
{'model': 'Mercedes-benz', 'price': 5000000}
```

Если это словарь то ?



`__dict__`

Добавим атрибут

```
class Car:
```

```
    def __init__(self, model, price):
        self.model = model
        self.price = price
```

```
    def __str__(self):
        return f"Модель: {self.model} с ценой {self.price}"
```

```
obj = Car("Mercedes-benz", 5_000_000)
```

```
obj.__dict__['color'] = "red"
```

```
print(obj.__dict__)
```



`__dict__`

Когда мы создаем объект класса, атрибуты этого объекта сохраняются в словарь под названием `__dict__`.

```
class Article:
    def __init__(self, date, writer):
        self.date = date
        self.writer = writer

article = Article("2020-06-01", "xiaoxu")
article.reviewer = "jojo"
print(article.__dict__)

{'date': '2020-06-01', 'writer': 'xiaoxu', 'reviewer': 'jojo'}
```

`__dict__` словарь для хранения атрибутов

Посмотрим атрибуты класса

```
class Car():
    "Класс Car"
    color = "Red"
    def __init__(self, model, price):
        self.model = model
        self.price = price

    def __str__(self):
        return f"Модель: {self.model} с ценой {self.price} "

print(Car.__dict__)

{'__module__': '__main__',
 '__doc__': 'Класс Car',
 'color': 'Red',
 '__init__': <function Car.__init__ at 0x7f74c5c9d790>,
 '__str__': <function Car.__str__ at 0x7f74c5c9d8b0>,
 '__dict__': <attribute '__dict__' of 'Car' objects>,
 '__weakref__': <attribute '__weakref__' of 'Car' objects>}
```



`__slots__` ограничение атрибутов

Определив волшебный метод `__slots__` мы ограничим кол-во атрибутов для экземпляра класса.

```
class Article:
    __slots__ = ["date", "writer"]
    def __init__(self, date, writer):
        self.date = date
        self.writer = writer

article = Article("2020-06-01", "xiaoxu")
article.reviewer = "jojo" ← Что произойдет ?
print(article.__dict__) ← вызовет ошибку
```




Создание итераторов через методы `__iter__()` `__next__()`

Во многих современных языках программирования используют такие сущности как итераторы. Основное их назначение – это упрощение навигации по элементам объекта, который, как правило, представляет собой некоторую коллекцию (список, словарь и т.п.). Язык Python, в этом случае, не исключение и в нем тоже есть поддержка итераторов. Итератор представляет собой объект перечислитель, который для данного объекта выдает следующий элемент, либо бросает исключение, если элементов больше нет.

Основное место использования итераторов – это цикл **for**.

`__next__` создание собственных итераторов

Если нужно обойти элементы внутри объекта вашего собственного класса, необходимо построить свой итератор. Создадим класс, объект которого будет итератором, выдающим определенное количество единиц, которое пользователь задает при создании объекта. Такой класс будет содержать конструктор, принимающий на вход количество единиц и метод `__next__`, без него экземпляры данного класса не будут итераторами.

```
class SimpleIterator:
    def __init__(self, limit):
        self.limit = limit
        self.counter = 0

    def __next__(self):
        if self.counter < self.limit:
            self.counter += 1
            return 1
        else:
            raise StopIteration
```

```
s_iter1 = SimpleIterator(3)
print(next(s_iter1))
print(next(s_iter1))
print(next(s_iter1))
print(next(s_iter1))
```



`__iter__` итераторов

Если мы хотим, чтобы с данным объектом можно было работать в цикле `for`, то в класс `SimpleIterator` нужно добавить метод `__iter__`, который возвращает итератор, в данном случае этот метод должен возвращать `self`.

```
class SimpleIterator:
    def __iter__(self):
        return self

    def __init__(self, limit):
        self.limit = limit
        self.counter = 0

    def __next__(self):
        if self.counter < self.limit:
            self.counter += 1
            return 1
        else:
            raise StopIteration

s_iter2 = SimpleIterator(5)
for i in s_iter2:
    print(i)
```



Заключение

- Магический метод — это специальные методы, которые вызывается неявно.
- Также это подход python к перегрузке операторов, позволяющий классам определять свое поведение в отношении операторов языка. Из этого можно заключить, что интерпретатор имеет "некую таблицу" соответствия операторов к методам класса. Перегружая эти методы, вы можете управлять "поведением" операторов языка относительно вашего класса.